Space-Economical Partial Gram Indices for Exact Substring Matching

Nan Tang
CWI, Amsterdam
The Netherlands
tang@cwi.nl

Lefteris Sidirourgos
CWI, Amsterdam
The Netherlands
lsidir@cwi.nl

Peter Boncz
CWI, Amsterdam
The Netherlands
boncz@cwi.nl

ABSTRACT

Exact substring matching queries on large data collections can be answered using *q-gram* indices, that store for each occurring q-byte pattern an (ordered) *posting list* with the positions of all occurrences. Such gram indices are known to provide fast query response time and to allow the index to be created quickly even on huge disk-based datasets. Their main drawback is relatively large storage space, that is a constant multiple (typically > 2) of the original data size, even when compression is used. In this work, we study methods to conserve the scalable creation time and efficient exact substring query properties of gram indices, while reducing storage space. To this end, we first propose a *partial gram* index based on a reduction from the problem of omitting indexed q-grams to the *set cover problem*. While this method is successful in reducing the size of the index, it generates false positives at query time, reducing efficiency. We then increase the accuracy of partial grams by splitting posting lists of frequent grams in a frequency-tuned set of *signatures* that take the bytes surrounding the grams into account. The resulting *qs-gram* scheme is tested on huge collections (up to 426GB) and is shown to achieve an almost 1:1 data:index size, and query performance even faster than normal gram methods, thanks to the reduced size and access cost.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Miscellaneous

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

Finding all instances of an ordered sequence of bytes (i.e., a string) in a large file is a fundamental pattern matching problem. Efficient solutions with real-time performance are attractive to many applications, e.g., online string search, word processing software, computational biology and digital forensic pattern search. Both the database and IR communities have investigated many string search topics, for instance approximate (e.g., edit distance based) string selection and top-K joins [1, 10, 12, 20], full text search [2, 13], and digital forensic search [18]. In this work we stick to the basic problem of exact substring matching [14, 16], but note that techniques for solving this are typically the building blocks to address elaborate approximate string matching problems.

		suffix array	full qgram	partial gram	**qs gram**
creation:	time	– –	++	++	+
	space	– –	–	++	+
querying:	short (≤ 5)	+	+	–	–
	medium	+	+	–	+
	long (> 10)	++	+	+	++
result-use:	merge	– –	+	+	+

Table 1: Exact Substring Index Comparison

We focus on the following efficiency aspects of the exact substring search indexing problem:

- index creation time,
- index space, and
- query response time.

These aspects are influenced by the following dimensions:

- data size (scalability),
- data distribution, in particular the frequency distribution of co-located letters,
- the length of the queries.[1]

Ideally, one would like to achieve an index that takes similar space (or less) as the input data, can be created in time close to a sequential I/O pass over that data, and provides query response time close to a sequential I/O pass over the query result volume; with any data and query distribution. These objectives are very hard to achieve *all together*, and get even harder when considering datasets with a uniform rather than a skewed distribution.

Suffix Techniques. In the algorithms community, the suffix tree [8, 16] and suffix array [9, 14] are being studied as indices to allow fast substring searching. Due to the large amount of information carried in each node and edge of a suffix tree, the storage overhead is 10-20x the input data in good implementations, which renders suffix trees impractical in most applications. The suffix array is a more space efficient variant which simply constructs an array of positions, where the positions point to the suffixes in the string in lexicographical order. For both suffix trees as well as

[1]We assume a distribution of co-located query letters similar to the data distribution.

CIKM'09, November 2–6, 2009, Hong Kong, China.

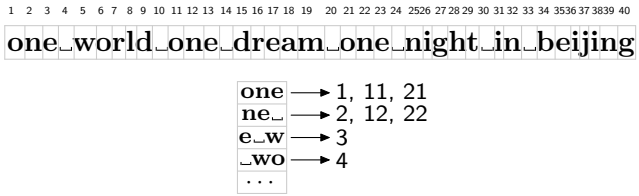


Figure 1: A String and Its Posting Lists

suffix arrays, algorithms are known whose space and time complexity are $O(|\sigma|)$, where $|\sigma|$ is the length of input data string σ. The excellent property of suffix arrays is that regardless the query (long or short, frequent or rare), lookup is simply a binary search of fixed cost. Sometimes this binary search, which leads to $log|\sigma|$ (on string σ) I/Os, is held against suffix arrays in favour of potentially O(1) hash-based solutions. However, this disadvantage can be mitigated by creating a sparse RAM resident B-tree index on top of the array. Regrettably, however, all known suffix array construction algorithms do *not* truly scale to large datasets. Once the data size exceeds main memory, index creation performance strongly deteriorates due to the need for close to $|\sigma|$ random disk writes. As a result, the best scaling reported "linear" suffix array construction algorithm (using special Linux kernel patches to improve random I/O) has been demonstrated on only 5GB of data [9]. Even on this effectively small – potentially RAM resident – data size, suffix array construction takes many hours. Our work focuses on truly scalable techniques that can be used to manage huge disk-based datasets, where the input data set can – in the case of forensic data search – be a set of full hard drive images (terabyte scale and beyond). As the scalability aspect of suffix methods is several orders of magnitude off target for our objectives, we focus on alternative methods.

Gram Techniques. The basic idea of q-grams is to construct an index on all occurring patterns of q co-located letters (we stick to bytes, in this work – depending on the alphabet a byte can either contain more or less than one letter). Figure 1 shows a string for a slogan of the Beijing 2008 Olympic games. Generally speaking, the whole string is indexed using q-grams, with each q-gram associated with a list of postings e.g. 3-gram postings as shown in Figure 1. Postings can be stored using IR engines specialising in inverted list processing, but it has been shown that a DBMS can also be used to store and query such data efficiently [10]. A given query ϱ (for exact match) is also decomposed to several q-grams $\mathcal{G}$. By performing merge join on the posting lists associated with each q-gram in $\mathcal{G}$, we can identify all occurrences of ϱ. For example, to find a phrase with the first word 'one' and the second word starting with 'w', we can submit a 5-byte query 'one␣w'. By doing merge join of the three posting lists of 'one' $(1, 11, 21)$, 'ne␣' $(2, 12, 22$ and 'e␣w' (3), we could identify one match at posting 1. IR systems routinely optimize posting list processing by choosing a merge join that puts the shortest lists first. Also, it is not strictly required to join the posting lists of *all* occurring grams; posting lists may be pruned (omitted from the plan) as long as all letters in the query remain covered by at least one non-pruned gram. A typical approach in gram query processing is thus to order the shortest posting lists first in the plan and prune the longest lists as long as the query remains covered.

Comparison. The creation of a gram-based (inverted) index boils down to using a fast scalable sort method, meeting our scalability objectives. As for the querying aspect, the query processing complexity is linear in the volume of all accessed (non-pruned) posting lists. While this can be substantially more than the final result volume, the positive point is that the access pattern is sequential, thus efficient. In text datasets, there is skew in the q-gram distribution, which results in long posting lists being pruned from query plans, strongly improving query time. The main disadvantage of q-gram based indices is storage space: (i) each byte is stored q times in index entries, e.g., the byte 'e' is stored three times at 'one', 'ne␣' and 'e␣w'; (ii) each byte is stored as a posting, and in case of terabyte datasets one needs 6 bytes for each. Posting lists consist of monotonic increasing numbers, hence these are routinely stored as differences (gaps) and compressed [3, 11]. Skewed data distributions compress well, giving a >2x space reduction. Even with compression, gram indices have size of 2-3x the input in case of skewed data. In more uniform data such as hard disk dumps with many binary video files, there are orders of magnitude more different q-grams (see Table 2), and most posting lists are short and thus cannot be compressed, such that the storage space deteriorates to 6x (additionally, the gain of pruning lists during query processing is much less). Note that suffix arrays, which consist of a fully permuted array of postings, are by definition not compressible, hence also take 6x space. Another drawback of suffix arrays is that when the result of a query is used in complex query processing, e.g., in case of regexp string matching, or when in XML databases keyword constraints are combined with structural constraints, multiple results need to be merged. Given the order in gram indices, merging is cheap, whereas suffix arrays need to re-sort the result.

	3-gram	4-gram	5-gram	6-gram
wikipedia 2.1GB	2.6×10^5	2.0×10^6	8.2×10^6	2.2×10^7
INEX 4.5GB	4.4×10^5	3.7×10^6	1.5×10^7	4.1×10^7
aquaint 3.0GB	1.8×10^5	1.4×10^6	7.0×10^6	2.2×10^7
XMark 7.4GB	4.9×10^4	4.0×10^5	1.8×10^6	5.9×10^6
movie.avi 699MB	4.2×10^6	7.9×10^7	8.9×10^7	9.0×10^7
movie.rmvb 1.2GB	4.2×10^6	1.3×10^8	1.5×10^8	1.5×10^8

Table 2: q-gram statistics for various datasets

Table 1 summarises the strengths and benefits of the discussed approaches, and also introduces the qs-gram method proposed in this paper. The primary purpose of the qs-grams is to reduce the size of the index further, close to a 1:1 relationship with the input size, while conserving the other good properties of gram indices.

Contributions and Roadmap. The problem of gram indices for substring matching is formalised in Section 2. In Section 3 we investigate the possibility of omitting certain grams from the index ("partial grams"), mapping the gram selection problem to a *weighted set covering problem*. For example, in Figure 1, in the partial gram approach, we might index 'one', 'e␣w', ⋯ (omitting 'ne␣'). A consequence of the partial gram storage is that we now get false positives of string occurrences that may not fully match the beginning and ending letters of the query, affecting performance. To reduce false positives, we then introduce the qs-gram structure, that adaptively augments the partial grams with signatures covering adjacent bytes. Section 4 describes implementation details. The performance is studied in Section 5, covering both text and binary data, with sizes up to 426GB (the largest experiment reported in literature so far)

showing that the *qs*-gram approach combines good query performance with strongly reduced index space. After discussing related work in Section 6, we conclude in Section 7.

2. PRELIMINARIES

A *string* σ is defined to be an ordered sequence of bytes, and its *length* to be $|\sigma| = n$. We denote $\sigma[i, j], 1 \leq i \leq j \leq n$ the substring of σ of length $j - i + 1$ starting at position i. If $i = j$ then we write $\sigma[i]$.

A *q-gram* is a string of length q. A q-gram g *appears* at position i of the string σ, if $g = \sigma[i, i + q - 1]$. The set of all q-grams that appear in a string σ is denoted by $\mathcal{G} = \{g_1, \ldots, g_k\}$. Each q-gram $g_i \in \mathcal{G}$ is associated with a list of *postings* $\mathcal{P}(g_i) = \{p_1, \ldots, p_l\}$, where each *posting* $p_i \in \{1, \ldots, n - q + 1\}, i \in \{1, \ldots, l\}$ refers to a position in the string σ where g_i appears. Evidently, a q-gram g_i can appear more than once in a string σ.

A q-gram g *covers* the bytes $\sigma[i], \ldots, \sigma[i + q - 1]$ if there is a posting $p \in \mathcal{P}(g)$ such that $p = i$. A set $\mathcal{G}$ of q-grams *covers* the entire string σ, if every byte $\sigma[i]$ of the string σ is covered by at least one q-gram.

Finally, a *string matching query* is defined to be the query which given the string ϱ of length $|\varrho|$, requests all positions i of the string σ such that $\varrho = \sigma[i, i + |\varrho| - 1]$.

Q-Gram based indices utilise the set of q-grams $\mathcal{G}$ to efficiently answer queries on approximate and exact substring match, text auto-completion, and for error-correction. The general approach, which will be referred to in the sequel as the *full q-gram index (FG)*, covers each byte of the string σ exactly q times. This is achieved by including in the set $\mathcal{G}$ all possible q-grams of the string σ, thus each byte $\sigma[i]$ will be covered by those q-grams that have postings equal to $i - q + 1, \ldots, i$. The most common choice for q is 3. The total number of postings in a full q-gram index is $|\sigma| - q + 1$.

In this work, we argue that for *space-economical* q-gram based indices it is better to use *partial q-gram indices (PG)* instead of full. However, for providing complete answers to substring matching queries, we have to ensure that each byte in the string σ is covered at least once. The problem at hand is formally defined as follows: *Given a string σ and its set $\mathcal{G}$ of q-grams, find a minimal subset $\mathcal{G}' \subseteq \mathcal{G}$, such that $\mathcal{G}'$ covers the entire string σ.* The *coverage* requirement is necessary to ensure that no matching substrings are missed, thus resulting in false negatives. However, a partial q-gram index will return exact matches plus a small list of candidate substrings because only a subset of $\mathcal{G}$ is indexed. In Section 5 we experimentally compare the number of candidates and the actual matches for the partial q-gram indices.

Another important difference between a full q-gram index and a partial one is that the former can match any substring that is no shorter than q, while the latter loses some expressive power: it can find matching candidates for a substring that is no shorter than $2q - 1$.

LEMMA 1. *If $\mathcal{G}'$ is a partial set of q-grams for string σ and ϱ is a* query *on σ that can be evaluated correctly, then the length $|\varrho|$ of ϱ is no shorter than* $2q - 1$.

PROOF. There are 3 cases. First, if $|\varrho| < q$ then it is impossible to match any q-gram with the query string ϱ since the length of ϱ is smaller than the length of the indexed q-grams[2]. The second case is when $q \leq |\varrho| \leq 2q - 2$. Let query ϱ match on string σ at positions $\sigma[i, i + 2q - 3]$. Also, assume that these bytes of σ are covered by the two q-grams g_1, g_2 with postings $\mathcal{P}(g_1) = \sigma[i - 1, i + q - 1]$ and $\mathcal{P}(g_2) = \sigma[i + q, i + 2q - 2]$. In this case, no q-gram that can be extracted from query ϱ will match any q-gram used to cover the bytes of string σ from position i to $i + 2q - 3$, and thus false negatives will appear. Finally, the third case is when $|\varrho| \geq 2q - 1$, then there should always be a q-gram that appears on both the query ϱ and the string σ. If no such q-gram exists, then the query result is empty. □

[2]We discount the non-practical approach of merging all superset grams $\{P_x | x \in \mathcal{G}' : x \subset q\}$ into a list of candidates.

Positional Merge Join. Consider two q-grams g_1, g_2 and their respective postings $\mathcal{P}(g_1) = \{p_1, \ldots, p_l\}$ and $\mathcal{P}(g_2) = \{p'_1, \ldots, p'_n\}$ over string σ. Also, consider a query ϱ where the q-gram g_1 appears in position i of ϱ, and q-gram g_2 in position $i + k$ of ϱ, where $i + k < |\varrho| - q + 1$. In order to find which postings of g_1 and g_2 are valid candidates for matching query ϱ to string σ, we have to join $\mathcal{P}(g_1)$ with $\mathcal{P}(g_2)$. This join must take into consideration also the distance k between g_1 and g_2 in ϱ because the same distance must be preserved in σ, too. The *positional merge join* is the join operation of two posting lists and an offset k.

For the two posting lists $\mathcal{P}(g_1)$, $\mathcal{P}(g_2)$ and an offset k, the operation $PosMergeJoin(\mathcal{P}(g_1), \mathcal{P}(g_2), k)$ is defined as:

SELECT $\mathcal{P}(g_1).posting$
FROM $\mathcal{P}(g_1).posting, \mathcal{P}(g_2).posting$
WHERE $\mathcal{P}(g_1).posting = \mathcal{P}(g_2).posting - k$

Example 2.1: *Let assume that there is a partial 3-gram index for the string in Figure 1. This partial index includes the 3-gram 'one' with postings list $\mathcal{P}$('one') = {1, 11, 21} and the 3-gram 'e␣w' with postings list $\mathcal{P}$('e␣w') = {3}. Let the query ϱ ='one␣w'. Query ϱ is decomposed to three 3-grams 'one', 'ne␣', and 'e␣w'. Since only 3-gram 'one' and 'e␣w' exist in the partial index, we apply the positional merge join operator on the postings lists of these two 3-grams with an offset 2. The result is posting {1}, which is the position that ϱ matches the string in Figure 1.*

The positional merge join will return a candidate list of positions in the string σ for the query ϱ to be verified. This happens when there is no q-gram in the index that covers the first and last $q-1$ bytes of the query ϱ. However, all bytes in between should be matched with some q-grams in the partial index. If this is not the case, then it is safe to conclude that there is no occurrences of query ϱ in σ. This provides opportunity during query evaluation, to reject mismatches early, and to only verify candidates in the raw data (i.e., the string σ) at the very end, only for those queries whose borders were not fully covered.

3. PARTIAL Q-GRAM INDICES

This section presents techniques for choosing the appropriate set of q-grams to be included in a partial q-gram index. To this end, we employ existing techniques from the set-theory field and extend them to meet practical requirements in terms of indexing and querying time. In addition, we introduce *signature* and *hash based* q-grams and then propose a novel *qs*-gram approach.

Partial Q-Gram Selection. The selection of the q-grams to be included in the partial index should satisfy the following two objectives: *i)* each q-gram must have a sorted list of postings in order to minimise I/O during posting fetching,

```
input  : σ = the input string, q = the length of the grams
output: a set of q-grams G' whose postings cover σ
1  G' ← ∅;
2  scan σ and derive all posting lists P(g_i), g_i ∈ G;
3  calculate F(g_i) = |P(g_i)| for all g_i ∈ G;
4  sort all F(g_i) in descending order;
5  cnt[i] ← 0 for i ∈ {1, ..., |σ|};
6  for each g_i in descending order of F(g_i) do
7      selected ← false;
8      if g_i is the first or the last q-gram in σ then
9          G' ← G' ∪ {g_i};
10         selected ← true;
11     else
12         for each posting p in P(g_i) and selected is false do
13             for k from 0 to q − 1 do
14                 if cnt[p + k] = q − 1 then
15                     selected ← true;
16                     G' ← G' ∪ {g_i};
17                     break;
18
19     for each posting p in P(g_i) do
20         for k from 0 to q − 1 do
21             if selected then
22                 cnt[p + k] ← q;
23             else if cnt[p + k] < q − 1 then
24                 cnt[p + k] ← cnt[p + k] + 1;
25
26 return G';
```

Algorithm 1. ChoosePartialGrams(σ, q)

and *ii)* the final candidate list must be small enough to reduce the cost of verification against the raw data, i.e., the input string σ. These two requirements converge to index only highly selective q-grams, i.e., the ones with a small number of postings. More formally, we minimise the function $\sum_{g_i \in \mathcal{G}'} \mathcal{F}(g_i)$, where $\mathcal{F}(g_i)$ denotes the frequency of q-gram g_i, i.e., the number of postings in the list $\mathcal{P}(g_i)$. This problem can be reduced to the *weighted set covering problem*. In terms of the set covering problem, the universe to be covered is all positions of the string σ, i.e., all $p_i \in S = \{1, \cdots, n\}$, where n is the length of σ. The collection of all posting lists $\mathcal{P}(g_i)$ of the q-grams g_i appearing in σ are used to cover the universe $S = \{1, \cdots, n\}$. The weight of each $\mathcal{P}(g_i)$ is the number of elements in the list, i.e., $\mathcal{F}(g_i)$.

The weighted set covering problem is a well studied problem and the optimal solution can be approximated by the following greedy algorithm, within an approximation factor of $H_n (\leq \ln n + 1)$. Let $\gamma(g_i) = \mathcal{F}(g_i)/|\mathcal{P}(g_i) \cap S|$ be a weight function, which considers both the q-gram frequency $\mathcal{F}(g_i)$ and the number of postings $|\mathcal{P}(g_i) \cap S|$ to be covered next. At each step, the q-gram g_i with the minimum value of the weighted function $\gamma(g_i)$ is picked and added to the partial index. Next, all postings that the posting list $\mathcal{P}(g_i)$ could cover are removed from the set S. Set S contains all the uncovered positions up until this point. Finally, the function γ is re-computed for all remaining unused q-grams. The process is repeated by choosing the next smallest weighted q-gram, until all positions are covered, i.e., until $S = \emptyset$.

The greedy algorithm produces a good approximation of the optimal solution in polynomial – to the size of σ and number of posting lists – time. However, the complexity remains quadratic, thus when both n and $\mathcal{G}$ are large, the classical greedy algorithm fails: it already takes hours to select 3-grams from a 64MB document.

3.1 A Scalable Set-Cover Algorithm

The complexity of the previous greedy algorithm originates from the requirement that in each step the *new global minimum* value of $\gamma(g_i)$ is selected. The basic idea of Algorithm 1 is to scan all q-grams in descending frequency order and greedily choose which grams to keep without recomputing the global weights in each step. More specifically, a q-gram g_i is *not* included in the partial index if it has a large frequency $\mathcal{F}(g_i)$, and only if all positions can be still covered by the remaining q-grams.

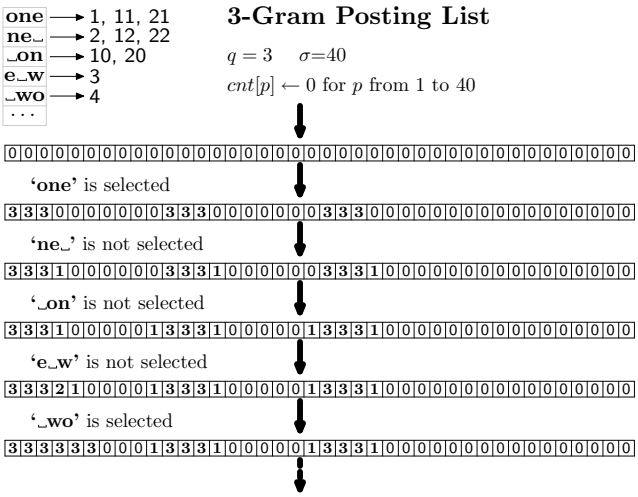


Figure 2: Algorithm Sample

Example 3.1: *Going back to the example in Figure 1, the string σ = 'one␣world␣one␣dream␣one␣night␣in␣beijing' has 40 positions to be covered and 33 distinct 3-grams. A single pass over string σ (line 2) will extract all q-grams and their posting lists. The q-grams are then sorted in descending order of their frequencies (line 4). Next, a counter cnt[i] for each position in the string σ is initialized to 0 (line 5).*

Figure 2 depicts the first few steps of the ChoosePartialGrams *algorithm for the string σ of Figure 1. The most frequent q-gram is 'one', but it has to be selected because it covers the first position of string σ. Thus, $\mathcal{G}' = \{$'one'$\}$ (lines 8-10). Next, the count* cnt *of each position in string σ covered by 'one' is set to 3. The second row of Figure 2 depicts the values of the count array* cnt *for each position in the string σ after 3-gram 'one' has been selected.*

The second most frequent 3-gram is 'ne␣'. The positions that 3-gram 'ne␣' covers are either already covered (e.g., position 2) or the count value cnt *is less than q-1. Therefore, 3-gram 'ne␣' can safely be disregarded since it is too frequent and other 3-grams do exist for covering the uncovered positions. However, before disregarding 3-gram 'ne␣', each of the positions $\{4, 14, 24\}$ is incremented by 1 to signify that there are still q-1 q-grams left that can potentially cover these positions. The third row of Figure 2 depicts the new state of the count array.*

The next two 3-grams are 'ne␣' and 'e␣w'. Both of them can be disregarded, because they are high in the frequency list and none of the positions that they cover on string σ have a count cnt *equal to q-1. However, '␣wo' has to be selected to ensure that position 4 of string σ is covered.*

ChoosePartialGrams *terminates with thirteen 3-grams selected for the partial 3-gram index: $\mathcal{G}' = \{$one, ␣wo, rld, d␣o, ␣dr, eam, m␣o, ␣ni, ght, ␣in, ␣be, iji, ing$\}$.*

The complexity of algorithm ChoosePartialGrams is $O(|\sigma| + |\mathcal{G}|ln|\mathcal{G}|)$. First, a linear scan is performed on σ to derive all q-grams, their postings lists, and the frequencies

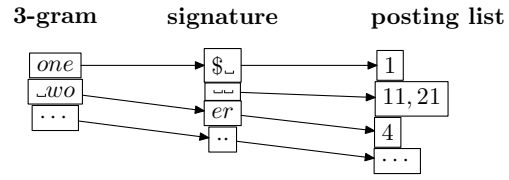


Figure 3: Signature-based Posting List

$\mathcal{F}(g_i)$ (lines 2-3). The cost of sorting all q-grams (line 4) is $O(|\mathcal{G}|ln|\mathcal{G}|)$. The big loop (lines 6-26) is a linear in the size of σ, since each position of σ will be visited at most q times.

3.2 Signature-based Gram Indices

There are two main limitations regarding partial q-gram indices: *i)* some very frequent q-grams are unavoidably selected to ensure that each byte of the string σ is covered; *ii)* during query evaluation, the indexed partial q-grams might not be sufficient to fully answer a query ϱ, thus resulting in a candidate list. In order to prune the false positives from the candidate list, the string σ has to be examined, causing time consuming I/O access.

To alleviate the aforementioned limitations, we propose the use of *signatures*. A signature s of a q-gram g is defined to be the concatenation of the two bytes *guarding* g in string σ. More specifically, let a q-gram be g_i and its posting list be $\mathcal{P}(g_i) = \{p\}$. Then, the signature s of g_i in position p is $s = \sigma[p-1]\sigma[p+q]$. A q-gram may have as many signatures as the number of postings in its posting list $\mathcal{P}$. The collection of all the signatures of a q-gram g, together with its posting list will be referred to as an s-gram. The q-gram g is also called the *infix* of its s-gram. Finally, we will refer to the first byte of a signature s as $s[1]$ and the second as $s[2]$. If $s[1]$ or $s[2]$ do not correspond to a byte (i.e., they point to the positions before the start or after the end of the string σ) then the symbol \$ is used.

Example 3.2: *Figure 3 shows the signatures s of the first two q-grams of the string σ of Figure 1. The 3-gram 'one' has the posting list $\{1, 11, 21\}$ and the list of signatures $\{\$␣,␣␣\}$. Consequently, the postings list of 3-gram 'one' can be split to two smaller ones, namely $\{1\}$ and $\{11, 21\}$, each one referring to a different signature of 'one'.*

Signatures are used to split long posting lists, in order to reduce both the number of postings fetched and the number of candidates produced during query processing. This is true because s-grams are more discriminative than q-grams. However, signatures may have a considerable storage overhead. In theory, the number of signatures for each q-gram is at most 2^{16} (for 2 bytes), but in practice the number of signatures for most q-grams is far below this upper bound.

The frequency distribution of q-grams in the text data sets we used loosely follows Zipf's law. This indicates that many q-grams will have short posting lists. For these q-grams, the overhead of maintaining their signatures is higher than retrieving the whole posting list during query evaluation. Moreover, some long posting lists may be split to very fine grained pieces, as a result of a large number of different signatures. For these cases, it is preferable to combine the fine grained lists into groups and thus save storage overhead. The next section presents techniques on how to dynamically make the decision on which posting lists to keep unmodified, which ones to split with the use of signatures and which ones to be combined into groups.

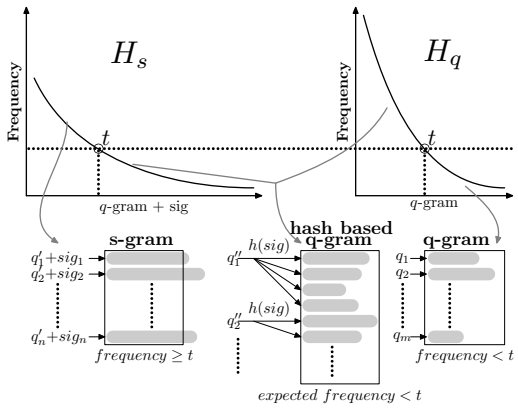


Figure 4: An Overview of QS-Gram

QS-Grams. We propose a novel approach to marry the merits of q-gram and s-gram, named qs-gram, with the following objectives: maintain less grams on the index entries, split long posting lists for improved query performance, and merge short postings for better compression ratio.

Figure 4 gives an overview of qs-gram. Two histograms are required, H_s and H_q, for s-grams and q-grams, respectively. Assume that $q = 3$, the number of distinct q-grams is up to 2^{24} and for s-grams 2^{40}. The histogram H_s might be too large to fit in memory. A *lossy* technique will be discussed in Section 4 on how these histograms can be efficiently built in the secondary memory.

As depicted in Figure 4, there are less q-grams than s-grams, and in average, the frequency of a q-gram is higher than that of an s-gram. Suppose that a threshold t is used to determine whether a gram is frequent or not, the grams are classified as follows:

1. Frequent s-grams ($frequency \geq t$) will end up with a private posting list. Referring to the left part of H_s in Figure 4, they reside in an s-gram dictionary.
2. Infrequent q-grams ($frequency < t$) will not be split at all. These q-grams are maintained in a q-gram dictionary, as shown in Figure 4, the right part of H_q.
3. For the grams in between, i.e., q-grams with frequencies no less than t and s-grams having frequencies less than t, a hash-based approach is utilised.

If a q-gram g has a long posting list, but has many s-grams with short posting lists, the hash function $h()$ is used to combine s-gram lists in a single one. To construct the hash-based q-gram dictionary, a number of buckets is dynamically allocated for each q-gram according to the frequencies of q-grams. For instance, if a q-gram g_e with frequency 35 is to be split and the threshold is $t = 10$, four buckets will be allocated for g_e. As the hash-function randomly combines infrequent signatures, the amount of postings falling in each hash bucket approximates t. The keys in the hash-based q-gram index are just q-grams stored in order, each leading to an array of buckets. To look up an infrequent s-gram, its infix q-gram g is first found, after which the hash function $h()$ is applied on s to compute the specific bucket, as shown in Figure 4. The main reason for using a hash function to combine buckets is that it requires no additional storage space.

The qs-gram index is designed to merge short posting lists of s-grams associated with one q-gram, so as to get almost equally length t posting lists. The certainty that posting

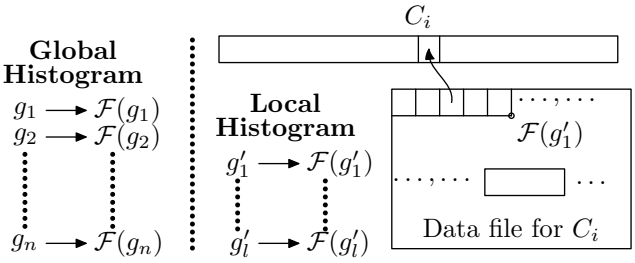


Figure 5: Global and local data structures

lists have a certain minimum length ensures that even in skewed gram distributions storage volume is dominated by postings, and not by the dictionary, and also ensures that compression is functional on all postings. As twice the list length t is an upper bound on the average number of false positives, it should be chosen relatively small.

4. IMPLEMENTATION DETAILS

This section describes the implementation details for building the q-gram indices. Since the input data is larger than the available memory, we present *chunk based* algorithms for full and partial q-gram indices that efficiently divide the process to fit in memory. We then present how a query is evaluated against the proposed q-gram indices.

4.1 Gram Generation

We describe not only effective, but also efficient and scalable generation algorithms for full q-gram, partial q-gram and qs-gram indices. Since, the input data may not fit entirely in main memory, and in order to achieve good scalability, a *chunk based* generation strategy is used. Specifically, the input data is partitioned into chunks, i.e., a sequence of bytes with equal length, and each chunk is processed independently. The algorithm can be executed either sequentially or in parallel over each chunk. After the termination of the chunk based generation, the *local* posting lists of each chunk are merged using a multi-way merge-union operation. The size of each chunk can be tuned, in order to render the algorithm cache or memory resident.

To decrease storage overhead, as well as to improve I/O system performance, all posting lists are compressed before being written to disk. Since each posting list is sorted, delta coding [3, 11] is adopted. Delta coding computes the difference between consecutive postings, and records only the value of the first posting and the following differences. Other compression techniques are orthogonally applicable.

For the chunk-wise processing, the input string σ is partitioned to $\lceil |\sigma|/m \rceil$ chunks $C_1, C_2, \ldots, C_{\lceil |\sigma|/m \rceil}$, each of size m. Each chunk C_i also includes $q-1$ bytes from the previous chunk C_{i-1} to ensure that no q-gram is missed during index creation. To illustrate this, assume that $q = 3$ and $m = 2$, thus string σ is partitioned into 2 chunks, C_1 and C_2:

$$\sigma = \sigma[1] \cdots \sigma[m-2] \underbrace{\sigma[m-1]\sigma[m]}_{q-1=2} \Big| \sigma[m+1]\sigma[m+2] \cdots \sigma[2m]$$

The first chunk C_1 is from position 1 until m, while the second chunk C_2, from position $m+1$ up to $2m$. However, this way each chunk is processed separately, and the 3-gram $\sigma[m-1]\sigma[m]\sigma[m+1]$ would never be found. To avoid this *miss*, C_2 is redefined to also include $q-1=2$ bytes from the previous chunk, thus C_2 starts at $m-1$ and ends at $2m$.

```
   input  : C_i = current chunk, q = gram length
   output: a local histogram H_i and a data file D_i
 1 k ← 0;
 2 define struct temp = {q-gram, position};
 3 while C_i[k] not the end of chunk do
 4 |   temp[k] ← C_i[k, k+q−1], k;
 5 |   k ← k+1;
 6 end
 7 radix_sort(temp) on q-gram, position;
 8 H_i ← merge the same grams and compute the frequencies;
 9 populate D_i from temp.position;
10 update global histogram with H_i;
11 return H_i, D_i;
```

Algorithm 2. CHUNK-WISE PROCESSING(C_i, q)

A *global* and a *local* histogram are used to record the frequencies of the q-grams found in string σ. The global histogram records the total frequency of each q-gram found on the *entire* input data. This histogram remains always in memory, since it is updated by every chunk. The left part of Figure 5 depicts the structure of the global histogram. It only consists of the q-grams and their frequencies.

On the other hand, a different local histogram exists for each chunk, which stores only the frequencies of the q-grams found in a specific chunk. To free up memory, each local histogram is flushed to disk after we finished processing the chunk. In addition to the local histogram, each chunk has its own *data file*, which contains all posting lists $\mathcal{P}$, i.e., the positions in the input string where the q-grams are located. The local histogram is used to navigate in the data file by computing the offset and the length for the posting list of each q-gram. The right part of Figure 5 depicts the structure of the local histogram and the data file for chunk C_i.

Algorithm 2 details how the local histogram and the data file is populated for each chunk. A temporary structure is used that records each q-gram and its position in the chunk C_i (line 2). First, chunk C_i is sequentially scanned and every q-gram is extracted and stored to *temp* (lines 3-6). Afterwards, the *temp* structure is sorted on the value of the q-grams by using the radix cluster algorithm [15] (a radix sort on the $8q$ lowest significant bits when viewing the concatenated bytes of a gram as a number) in $O(qm)$ time (line 7). Next, the *temp* structure is merged to contain all the postings of the same q-gram and the frequencies are computed (line 8). This is possible, because radix cluster will group all same q-grams together. Next, the data file $\mathcal{D}_i$ is populated by dumping sequentially the *merged* and *sorted temp* structure (line 9). Finally, the global histogram is updated by merging the current local histogram (line 10).

The above process is repeated for all chunks. Afterwards, the local data files $\mathcal{D}_i$ are merged to one single global data file $\mathcal{D}$. Each posting list of the local data files is written to the correct offset of the single global data file by consulting the global histogram. The resulting global data file contains the entire posting list of each q-gram in the input string σ.

An alternative approach is to scan the entire input string and collect the global histogram without writing data to disk. In a second scan, data could be written to the final global data file without merging local posting lists. However, this alternative approach writes data using random I/O, which renders the algorithm I/O bound. For the proposed approach, all disk reads/writes entail sequential I/Os, which is over an order of magnitude faster than random.

Index generation thus produces two global structures: a histogram H and a data file $\mathcal{D}$. The global histogram contains entries as $[q\text{-}gram|data\_offset]$, where *data_offset* refers

to the position in the data file that the q-grams's posting list can be found. The q-grams are sorted to facilitate binary search during query evaluation. The data file is the concatenation of the posting lists of all q-grams.

The generation procedure for partial q-grams is similar to full q-grams. The only difference is that in each chunk, instead of indexing all q-grams, partial q-grams are selected using the algorithm proposed in Section 3.1. The decision is made locally by consulting only the local histograms.

For generating the qs-gram index, two global histograms are required, as depicted in Figure 4. One global histogram is needed for q-grams and one for s-grams. These are used to assist the gram classification by examining their frequencies. The two global histograms are produced in the same way as described until now. However, the s-gram histogram is an array of [$(q+2)$-*gram, posting_list*] entries sorted first on the s-gram level then on the postings. Recall that the number of s-grams is up to 2^{40} when $q = 3$, which could render the s-gram histogram to be out of memory. A lossy *probabilistic counting* technique is adopted for this case, similar to [17], where there is a limited histogram buffer. In principle all s-grams are collected, but when the buffer gets full, s-grams with a small count are removed to make space. The adaptive approach works as follows. A lower bound, initially set to 1, is used to guard whether an s-gram should be removed or not. If after removing entries with only 1 occurrence, there is still not enough space, the lower bound is doubled and the above removal operation is repeated until there is enough space in the buffer. This leads to an incomplete histogram on the s-gram level, but generally only s-grams that occur very infrequently will be missing.

After the first scan of the input string, we determine using threshold t whether a q-gram should be split with signatures: *i*) very frequent s-grams will end up with a private posting list, recorded in an s-gram dictionary; *ii*) very infrequent q-grams will not be split at all, stored in a q-gram dictionary; *iii*) the hash based approach is used in between.

Next, the input string is scanned for the second time. In this second pass, the local histograms and data files are generated for each chunk. We point out two differences with full q-gram generation. First, for hash based q-grams, instead of maintaining for each q-gram a global offset, each bucket records an offset. Secondly, posting lists are sorted in s-gram level. Therefore, during the construction of the q-grams and hash based q-grams, the posting list is re-sorted.

The final multi-way merge-union on sorted posting lists is similar to that for q-grams. The only difference is that the three types of dictionaries must be handled separately.

4.2 Query Processing

Consider a query ϱ of size $|\varrho| = k$ where $k \geq 2q-1$. The query ϱ is decomposed into a set of q-grams $\mathcal{G}_\varrho = \{g_1, \ldots, g_{k-q+1}\}$, where each $g_i = \varrho[i] \cdots \varrho[i+q-1]$ for $1 \leq i \leq k-q+1$. If signatures are used, the signature s_i of each q-gram g_i is $s_i = \varrho[i-1]\varrho[i+q]$. The first byte s_1 of the first signature and the second byte s_{k-q+1} of the last signature are unknown. They are set to a wildcard $*$ that matches any single byte, i.e. $s_1[1] = s_{k-q+1}[2] = *$. With the above information about q-grams and signatures for the query ϱ, and the stored histograms, a set of posting lists is fetched from disk and the positional merge join is applied.

In an early optimization step, *negative queries*, i.e., queries that have empty results, can be quickly identified before even the postings are fetched. In the case of full q-grams, if any posting list of $g_i \in \mathcal{G}_\varrho$ is empty ($\exists \mathcal{P}(g_i) = \emptyset$), ϱ is a negative query, since a query will have an empty result if at least one of its substring does not appear. For example, let a string $\sigma_e =$'*one␣world*' and two queries ϱ_1 :'*one␣w*', 'ϱ_2 :*one␣v*'. For query ϱ_1, all its q-grams are indexed by the q-grams of σ_e. With regards to ϱ_2, one of its q-gram. i.e., $g_3 =$'e␣v', is not indexed, and thus ϱ_2 is a negative query over σ_e.

In the case of partial q-gram and qs-gram, the criteria are different since some q-grams are pruned: *i)* if all posting lists are empty ($\forall \mathcal{P}(g_i) = \emptyset$) where $g_i \in \mathcal{G}_\varrho$, ϱ is a negative query, and *ii)* if the i-th byte of the query ϱ is not covered by any q-gram, and $q-1 < i < |\varrho|-q+3$, then ϱ is a negative query. Notice that we can not determine if a query is negative if the bytes that are not covered are the $q-1$ bytes located at the borders of the given query.

The next step, and since it has been established that the query ϱ might have a non-empty result set, the relevant posting lists must be fetched. For the full q-gram index, the posting lists of all q-grams are indexed. However, in order to further optimize the query evaluation, q-grams with very long posting lists can be omitted during query evaluation. The basic requirement is that each byte of the query is covered at least by one q-gram. For instance, to cover a seven-byte query '*one␣wor*', the grams '*one*' and '*wor*' can be used. To cover the remaining byte '␣', any of the q possible grams (i.e., '*ne␣*', 'e␣w' and '␣*wo*') are checked, and the one with the smallest frequency is selected.

In the case of the partial q-gram index, unfortunately the same optimization techniques can not be applied. The q-grams are selected chunk-wise and based on local decisions (i.e., the local histogram), thus there is no guarantee that a specific q-gram will be found across the entire input string. Therefore, no matter whether a query ϱ is fully covered or not, all posting lists must be fetched and examined.

The qs-gram index has three histograms. The infrequent q-gram histogram is processed similarly to the partial q-gram index. However, the histograms for the s-grams and hash based q-grams are treated differently, since the stored signatures must be taken into account. First, the signature s_i of each gram g_i is extracted. There are three cases:

(1) $s_i = \varrho[i-1]\varrho[i+q]$, where $i \neq 1$ and $i \neq k-q+1$. The signatures are ordered, hence a binary search is used to identify the specific entry for s_i.

(2) $s_i = \varrho[k-q+1]*$, which refers to the last q-gram of query ϱ. With the use of binary search all signatures whose first byte is $\varrho[k-q+1]$ will be identified.

(3) $s_i = *\varrho[q+1]$ i.e., the first q-gram. The second byte of the signatures is not sorted, therefore a binary search can not be used and all signatures of g_i will be fetched.

If the q-gram is not found in the s-gram histogram, the hash based q-grams are searched. The hash value of s_i is calculated and used to identify the bucket that the q-gram should reside in. If it is found, the corresponding posting list is fetched. Notice that, if the signature contains a wildcard (for g_1 and g_{k-q+1}), the hash value cannot be decided and thus all posting lists of the current q-gram will be fetched. Finally, if the q-gram is not located either in the s-gram nor the hash based histogram, the q-gram histogram is queried.

All fetched relevant posting lists are then fed into a positional merge join operator in order to identify the valid

occurrences of the query ϱ in string σ. For the full q-gram index, a standard multi-way positional merge join is sufficient. However, for the partial q-gram and qs-gram indices, the positional merge join must adaptively determine in a chunk-wise fashion which q-grams are present and which were (potentially) pruned by the set-cover algorithm:

(1) When processing a chunk C_i, if all posting lists have no postings indexed in chunk C_i, or some byte $\varrho[k]$ of a query ϱ is not covered where $q-1 < k < |\varrho| - q + 3$, chunk C_i can be safely skipped,

(2) If some q-grams have postings (are indexed) in the chunk C_i, and the corresponding q-grams cover all bytes of the query[3], the standard multi-way positional merge join is used.

(3) Otherwise, that is, if the present q-grams cover the inner part of a query but not the outer boundaries, a multi-way positional merge join is still performed, but these results are marked as candidates, requiring verification afterwards.

Extensions. Possible future optimisations in qs-gram query processing address the problem that when examining a chunk and seeing that a certain q-gram is *not* present, we pessimistically assumed in the above that the q-gram must have been pruned, while it could also be that there was simply no occurrence (in that case, there is no query result in this chunk, and whatever we emit above is a false positive). To address this, we could augment our indexing structure with a bit-string for each q-gram, that records whether a q-gram was preserved by the set-cover algorithm in a specific chunk or not. For example, the bit-string 11001 means that a q-gram g is present in chunks $1, 2$ and 5. This information can then be used to emit considerable less false positives, but raises the future research question as to the space/time trade-offs of keeping this extra information, especially on data sets with many q-grams. Even if no bit-strings are kept, a similar optimization can already be made in the qs-gram approach when merging a signature or hashed list (i.e. in case of a globally frequent q-gram). If this list turns out to have no postings in a chunk, we can still analyze the other hash and signature tables for the same infix q-gram in this chunk. Because the set-covering pruning decisions are made on the q-gram level, presence of any other infix matching information in the indices is proof that the q-gram was not pruned in C_i, and can be used to reduce false positives. Note, however that searching for this extra proof may cause additional index I/O at query time. One potential strategy would be to use only a limited set of (or a single) long signature list(s) with the same infix to steer this optimization. Alternatively, when processing hash lists, with all hash-bucketed postings list adjacent on disk, we might simply re-use whatever information is present in the large disk block I/O unit we read for opportunistic pruning.

5. PERFORMANCE STUDY

We conducted extensive experiments with all discussed q-gram indices, investigating index creation time, index space and query performance. The following datasets were used:

- wiki-article: the Wikipedia archive that contains current versions of article content (22GB)[4].

[3]Note that if a certain q-gram is stored as an s-gram, we have information on $q+2$ query letters.

[4]http://download.wikimedia.org/enwiki/20090512

elapsed time in sec. (write time in parentheses)			
	FG	**PG**	**QS**
wiki-article 22GB	4165 (1495)	4274 (778)	9673 (882)
wiki-meta 44GB	8665 (3133)	8901 (1646)	20042 (2193)
XMark 55GB	11122 (4151)	10964 (2018)	24057 (2346)
single PC, two disks			
GOV2-1core 426GB	91364 (32365)	87045 (13475)	174872 (5507.9)
GOV2-8core 426GB	12539 (4966)	12068 (2555.4)	25129 (1018.37)
8-core server, 16 SSDs			

Table 3: Index Creation Performance (sec)

- wiki-meta: the complete Wikipedia archive including discussion and user pages (44GB).
- XMark[5], using scale factors from 10 to 500 to generate XML documents from 1.1GB to 55GB.
- GOV2: used in the 2006 TREC Terabyte Track [6] consisting of a crawl of the .gov domain (426GB).
- movie: we also experimented with a number of binary data files containing movies, and ranging in size from 1GB to 8GB, mainly to show the index creation and space behavior when there are many more different grams and the frequency is more uniform (we still lack application scenarios and queries that could be useful on such binary data, this is future work).

In the following, we use *FG*, *PG* and *QS* to denote full q-gram, partial q-gram and qs-gram indices. Most experiments were conducted on a PC with a 2.40GHz Intel Core2 Quad Q6600 CPU, 8GB of RAM and 2 hard disks in RAID-0. The experiments with the larger 426GB GOV2 dataset did not fit on the storage system of the PC, and were done instead on a server machine with two quad-core 2.8GHz Xeon (Nehalem) CPUs, 48GB of memory and a RAID-0 file system consisting of 16 SSDs (Intel X25-M). Note that in the index creation experiment, all I/O is sequential, and performance of SSDs is quite similar to normal hard drives. On this machine, we also performed a parallel experiment, where the 426GB dataset was split into eight partitions of 53GB, on which we ran our indexing program in parallel.

Index Creation. Table 3 shows the overall time needed to create the various indices, where we give below each result between parenthesis the sub-component time needed for the multi-way merge union. The chunk size was 128MB for *FG*, *PG* and *QS* (we used frequency threshold $t = 2000$), causing a first phase of index generation for a 128MB chunk of the input data at-a-time. For the subsequent of merging local posting lists within chunk into a global index, the buffer size was again set to 128MB, causing our external merge sort to read each time 128MB of postings from all chunks, writing this out as a global postings lists. All algorithms run CPU-bound. For *FG* and *PG*, we achieve an indexing speed of 18GB per hour on the PC, and on the server machine in the 8-way parallel experiment, the 426GB of GOV2 data was indexed in less than 3.5 hours, achieving good speedup.

For *QS*, the performance is around 8GB per hour on the PC. *SG* and *PG* require additional time for running the set covering algorithm, but in case of *PG* this extra cost is offset by the fact that it writes less postings (this can be seen by comparing the write times between parentheses). *QS* takes much more time, since (1) the radix-sort runs on $q+2$

[5]http://monetdb.cwi.nl/xml/

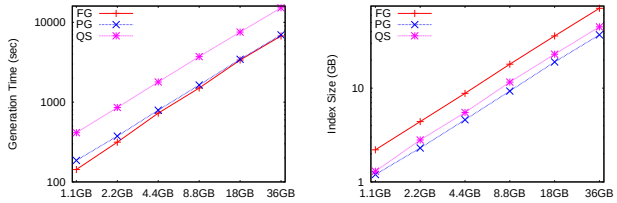
bytes instead of q bytes, (2) to save space without writing intermediate data, the raw string is scanned twice. The first scan is to collect two global dictionaries for s-gram and q-gram. The second scan processes chunk-wise data.

Index Space. Table 4 shows the sizes of generated indices, including the space for the dictionaries. The dictionary sizes depend on the amount of different grams (see Table 2). As the amount of different grams in the index is small in textual data, the dictionary sizes do not play any role in the results on the textual data sets in Table 4. Table 4 shows in the additional "dict:" row that for *QS* (the scheme with the largest dictionaries), this overhead is still limited. The overall conclusion for textual data is that *FG* with compression typically achieves a 2x storage space, and *PG* and *SG* improve that to roughly 1x. In case of binary data, index space deteriorates to 4x in *FG*. Both partial approaches perform similar, and reduce storage by almost a factor 2.

	wiki 22GB	wiki 44GB	XMark 55GB	GOV2 426GB	binary data (movie) 1GB	2GB	4GB	8GB
FG	43	89	111	606.7	3.9	7.6	16	31
PG	23	47	58	425.3	2.4	4.7	9.5	19
QS	23.7	50.7	69.7	490.3	2.6	4.9	9.7	19.8
dict:	0	0	0	0	.2	.2	.2	.3

Table 4: Index Storage Space (GB)

Figure 6 shows the scalability on XMark using sizes 1.1GB, up to 36GB. Figure 6 (a) confirms linear scalability in creation time and Figure 6 (b) confirms linear scalability in index sizes.



(a) Generation Time (b) Index Size

Figure 6: XMark data with factors 10, 20 to 320

Query Evaluation For each data set, queries with variable lengths were tested, from 5 to 15. For queries with the same length, 100 positive queries were randomly drawn from the dataset, but we excluded queries with excessively long results (> 1M). In the figures, the results are ordered on *FG* score to help comparison. Due to space limitations, we concentrate here on the 44GB wiki-meta data.

Figures 7 (a)-(d) shows that for short queries (e.g., 5), *FG* is generally faster than *PG* and *QS*. However, when the length of queries increases to 9, the difference between *FG* and *PG*, *QS* get smaller, and *QS* processes many queries faster than *FG*. For queries with lengths 11 and 15, *QS* outperforms *FG* for most queries (*PG* being slowest always).

To understand these results better, we first explain the other two groups of experiments. The first group is shown in Figures 7 (e)-(h), showing the number of postings loaded for different approaches. The number reported here was the posting under compression i.e. the data read from disk, but not the number of postings resulting from the positional merge join. The other group of experiments is the number of postings verified, which applies only to *PG* and *QS*, since they may generate false positives. These numbers are shown in Figures 7 (i)-(l).

Figures 7 (e)-(h) tell that *FG* and *PG* load many more postings than *QS*. The reason that *FG* sometimes loaded less data than *PG* is that *FG* was optimized to omit long posting lists from query plans. *PG*, however, cannot prune long posting lists, since that will potentially introduce many false positives. *QS*, however, always fetches less postings, since the posting lists of qs-grams are typically short.

The query evaluation for *FG* is CPU-bound, dominated by the number of postings in the positional merge join. The execution time of *PG* and *QS* is composed of two parts: (1) positional merge join, which is dominated by the number of postings loaded; (2) false positive checks, which will dominate the overall time if the number of candidates is large. Observe from Figures 7 (i)-(l) that *QS* always checks less candidates than *PG*. The value below 1 means that there is no candidates, i.e., all chunks are fully covered. When only few (or none) candidates are required to be checked, *QS* outperforms *FG* and *PG*. *PG* is normally slower than *FG*, since *PG* could load more postings and candidates are to be verified. Note that *PG* has the smallest storage space, roughly half of used by *FG*. *QS* needs slightly more space, but outperforms *PG* in query processing.

6. RELATED WORK

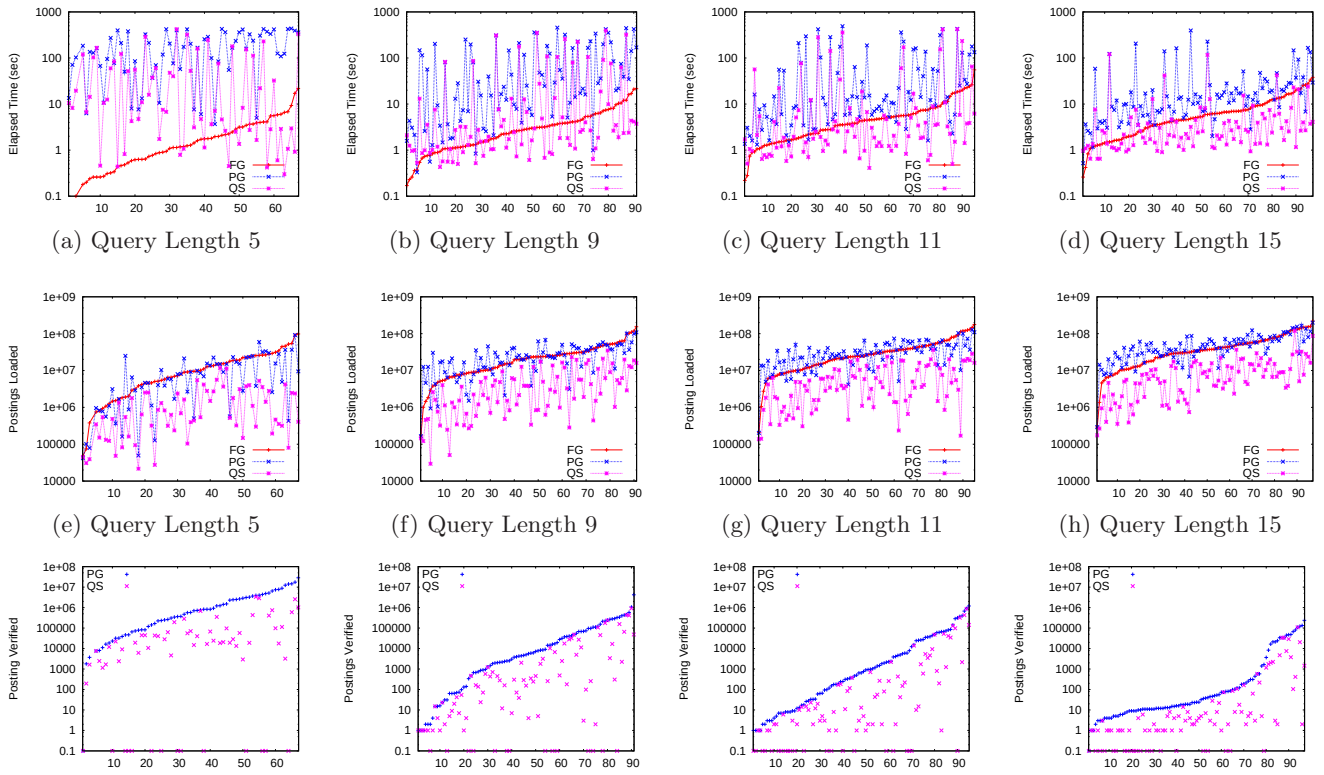
For answering exact substring matching, suffix tree [8, 16] and suffix array [9, 14] have been extensively studied, but this has not yet produced algorithms that could work on huge sizes such as the 426GB GOV2 dataset we indexed.

Q-gram based indices were first used to model sequences, using the statistical properties of q-grams. q-grams have been widely studied for efficient approximate string matching, also using DBMS as the storage and execution component [10]. In order to improve the performance of approximate queries, [12, 20] proposed variable-length grams, and [1, 5] worked on reducing the sizes of indices – note that our work concerns not approximate but with exact query processing. Many other work study the problem of approximate string joins using various similarity functions [4, 7, 19]. The above q-gram based approaches focus mainly on improving approximate string matching, and not robust enough to efficiently handle the problem of exact substring matching.

7. CONCLUSIONS AND FUTURE WORK

We presented different gram-based indices for exact substring matching on huge data sets. Motivated by the fact that the full q-gram index has considerable storage overhead, impacting particularly the ease of manipulating such huge data sets, we aimed at space economical q-gram indices. The proposed partial q-gram indices have a very compact size (around 1x), but produce a relative large number of false positives that must be checked against the raw data. To alleviate this problem in partial q-grams, we proposed a novel approach called qs-gram index. The qs-gram index is designed to exploit the gram distribution, by splitting long posting lists and merge short posting lists with a frequency-adaptive signature approach, to ensure good data compression and query performance. The trade-off compared to partial q-grams is that qs-grams consume slightly larger space. We demonstrated excellent scalability on large data sets (up to 426GB), and showed query performance of qs-grams rivals or even improves that of traditional full q-grams.

As to future work, to reduce the number of postings to



(a) Query Length 5 (b) Query Length 9 (c) Query Length 11 (d) Query Length 15

(e) Query Length 5 (f) Query Length 9 (g) Query Length 11 (h) Query Length 15

(i) Query Length 5 (j) Query Length 9 (k) Query Length 11 (l) Query Length 15

Figure 7: wiki-meta (44GB) (a-d) Elapsed Time (e-h) Postings Loaded (i-l) Postings Checked

be verified in query processing, we plan to build to a cost model to judge when is benefit to load and combine multiple posting lists intead of verifying the raw data. Also, the dictionary size (and auxiliary bit-strings) for large binary data is problematic. We will investigate dictionary compression techniques.

8. REFERENCES

[1] M.-S. K. 0002, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, 2005.

[2] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient xml search with complex full-text predicates. In *SIGMOD Conference*, 2006.

[3] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1), 2005.

[4] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[5] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, 2009.

[6] S. Büttcher, C. L. A. Clarke, and I. Soboroff. The trec 2006 terabyte track. In *TREC*, 2006.

[7] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, 2003.

[8] C.-F. Cheung, J. X. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Trans. Knowl. Data Eng.*, 17(1), 2005.

[9] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *J. Exp. Algorithmics*, 12, 2008.

[10] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-grams in a dbms for approximate string processing. *IEEE Data Eng. Bull.*, 24(4):28–34, 2001.

[11] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD Conference*, 2007.

[12] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, 2007.

[13] J. Lu and J. P. Callan. User modeling for full-text federated search in peer-to-peer networks. In *SIGIR*, 2006.

[14] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *SODA*, 1990.

[15] S. Manegold, P. A. Boncz, and N. Nes. Cache-conscious radix-decluster projections. In *VLDB*, 2004.

[16] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2), 1976.

[17] A. Metwally, D. Agrawal, and A. E. Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.*, 31(3):1095–1133, 2006.

[18] S. Petrovic and S. Bakke. Application of q-gram distance in digital forensic search. In *IWCF*, 2008.

[19] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, 2004.

[20] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*, 2008.